CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 6: Fundamental Algorithms

- Fundamental algorithms is about the methods to transform visualization data such as structured points and grids, unstructured grids, and polygonal data into graphics primitives.
6.1 Introduction

Algorithms of structural transformation:

- **Geometric transformation** (translation, rotation, scaling): geometric transformation changes the input geometry, but do not change the topology of the dataset.
- **Topological transformation**: topological transformation alters input topology but do not change geometry and attribute data. Such as converting a dataset type from polygonal data to unstructured grid data, or from structured points to unstructured grid.
- **Attribute transformation**: attribute transformation converts data attributes from one form to another, or create new attributes from the input data.
- **Combined transformations**: combined transformation changes both dataset structure and attribute data.

Algorithms according to the type of data they operate on or they generate:

- **Scalar algorithms**: operate on scalar data.
- **Vector algorithms**: operate on vector data.
- **Tensor algorithms**: operate on tensor matrices.
- **Modeling algorithms**: generate dataset topology or geometry, or surface normals or texture data. For example, generating glyphs oriented according to the vector direction and then scaled according to the scalar value, is a combined scalar/vector algorithms. Modeling algorithms means the algorithms does not fit squarely into one single category.
6.2 Scalar algorithms

- Color mapping
- Contouring
- Scalar generation
Color mapping

$ s_i < \min, i = 0$

$ s_i > \max, i = n - 1$

$i = n \left( \frac{s_i - \min}{\max - \min} \right)$

Color mapping is a common visualization technique that maps scalar data to colors and displays the colors on the computer system. The scalar mapping is implemented by indexing a color lookup table. Scalar values serve as indices into the lookup table.

The lookup table holds an array of colors (e.g., red, green, blue or other representations). A minimum and maximum scalar range (min, max) is associated with the table. Scalar values are mapped into the scalar range. Scalar values greater than the maximum range are represented by the maximum color, scalar value less than the minimum range are represented the minimum color.

The key to color mapping for scalar visualization is to choose the lookup table entries carefully.
Contouring

- Three-dimensional contours are called isosurfaces.
- Contouring always begins by selecting a scalar value, or contour value, that corresponds to the contour lines or surfaces generated. To generate contours, linear interpolation must be used.
- Two algorithms in contouring: marching cubes and edge-tracking.
- The marching cubes algorithm is easy to implement, but it creates disconnected line segments and points, and the corresponding merging operation requires extra computation resources.
- The edge-tracking algorithm generates a single polyline per contour line, avoiding the need to merge coincident points.
Marching squares (cubes) algorithms

- Select a cell
- Calculate the inside/outside state of each vertex of the cell
- Create an index by storing the binary state of each vertex in a separate bit
- Use the index to look up the topological state of the cell in a case table
- Calculate the contour location (via interpolation) for each edge in the case table
The basic assumption of the marching squares or marching cubes algorithms is that a contour can only pass through a cell in a finite number of ways. A case table is constructed that enumerates all possible topological state of a cell, given combinations of scalar values at the cell points. The number of topological states depends on the number of cell vertices, and the number of inside/outside relationships a vertex can have with respect to the contour value.

A vertex is considered inside a contour if its scalar value is larger than the scalar value of the contour line. Vertices with scalar values less than the contour value are said to be outside the contour.

If a cell has four vertices and each vertex can be either inside or outside the contour, there are $2^4=16$ possible ways that the contour passes through the cell. In the case table we are not interested in where the contour passes through the cell, just how it passes through the cell, topology of the contour in the cell.
Sixteen different marching squares cases. Dark vertices indicate scalar value is above contour value. Cases 5 and 10 are ambiguous.
Marching cubes cases for 3D isosurface generation. The 256 possible cases have been reduced to 15 cases using symmetry. Dark vertices are greater than the selected value. The six ambiguous cases are 3, 6, 7, 10, 12, 13.
Contouring ambiguity: contouring ambiguity happens on a 2D square or the face of 3D cube when adjacent edge points are in different states, but diagonal vertices are in the same state. Contouring means a cell can be contoured in more than one way.
Treating ambiguity in 2D

The case showing here corresponds to marching squares case 10. In two dimensions, for each ambiguity case, we implement one of the two possible cases: break and join. Either choice is acceptable since the resulting contour lines will be continuous and closed (or will end at the dataset boundary).
In 3D situation, we cannot simply choose an ambiguous case independent of all other ambiguous cases. In above figure the arbitrary chosen case 6c cause a hole in isosurface. There six carefully design complementary cases for the six ambiguous cases in 3D. No hole is found for the case that is designed to be compatible with neighboring cases to prevent the creation of holes.
Marching cubes
complementary cases

Case 3c

Case 6c

Case 7c

Case 10c

Case 12c

Case 13c
Scalar generation

\[ s_i = \frac{(p_i - p_l) \cdot (p_h - p_l)}{|p_h - p_l|^2} \]

Very often, the data may not be single-valued (i.e., scalar), or it may be a mathematical or other complex relationship. Scalar generation means we need to create a relationship between the data and the visualization techniques to generate a unique scalar value. One such example is to visualize the terrain data by color mapping or contouring, we can do this by coloring or contouring according to elevation, and need scalars to index the colormap. We can simply extract the z coordinate, and use the z-coordinate value as scalars. However, a method called normalized dot product is more general. Assuming \( p_l \) is the low point (sea level), \( p_h \) is the high point (mountain top), we can compute the normalized elevation scalar \( s_i \) at point \( p_i = (x_i, y_i, z_i) \).
6.3 Vector algorithms

- Vector data is a three-dimensional representation of direction and magnitude. Vector data often results from fluid flow or when examining derivatives (i.e., rate of change, flux) of some quantity.
Methods for vector visualization include oriented lines and glyphs, where glyphs mean any 2D or 3D geometric representation such as oriented triangle or cone. Vector visualization should represent both the magnitude and orientation of the data by the three components $(v_x, v_y, v_z)$. In the vector presentation, the line often begins with the Point with which the vector is associated, and the resulting line must be scaled up or down to control the size of its visual representation.
Warping

Vector data is often related with motion, such as fluid flow or displacement of a beam. This can be done by representing the displacement of a structure under load through deforming the structure, or creating a flow profile through distorting a straight line inserted perpendicular to the flow.

We must scale the vector field to control geometric distortion. Too small a distortion may not be visible, while too large a distortion can cause the structure to turn inside out or self-intersect.
Displacement plots

- Vector displacement on the surface of an object can be visualized with displacement plots. A displacement plot shows the motion of an object in the direction perpendicular to its surface. The object motion is caused by an applied vector field.

- The idea of vector displacement plots comes from the observation that vectors are converted to scalars by computing the dot product between the surface normal and vector at each point. If positive values result, the motion at the point is in the direction of the surface normal (i.e., positive displacement). Negative values indicate that the motion is opposite the surface normal (i.e., negative displacement).

\[ s = v \cdot n \]

Scalar computation
\[ dx = v \cdot dt \]

\[ x(t) = \int_{t_0}^{t} v \, dt \]

The integral of the differential equation, where \( x(t) \) is the position at time \( t \).

The accuracy of the numerical integration is a function of the step size \( dt \). To visualize the same motion of a bubble, the intermediate steps may not be seen using a large time step. Sometimes the solution is totally wrong if a large time step is used.
Numerical integration: Euler method

\[ x_{i+1} = x_i + v_i \Delta t \]

Taylor expansion for \( x(t) \) about \( t = t_i \)

\[ x(t_i + \Delta t) = x(t_i) + \Delta t x'(t_i) + \frac{\Delta t^2}{2} x''(t_i) + ... \]

\[ = x_i + \Delta t v_i + \frac{\Delta t^2}{2} x''(t_i) + ... \]

Truncation error

\[ E = O(\Delta t^2) \]

Euler method is first order in accuracy. In general, an integration method is called \( n^{th} \), if the truncation error per step is \( O(\Delta t^{n+1}) \)
Numerical integration: Runge-Kutta method

\[ v(t, x) = x', x(t_i) = x_i \]

\[ x_{i+1} = x_i + \frac{\Delta t}{6}(k_1 + k_2 + k_3 + k_4) \]

\[ k_1 = v(t_i, x_i) \]

\[ k_2 = v\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_1\right) \]

\[ k_3 = v\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_2\right) \]

\[ k_4 = v(t_i + \Delta t, x_i + \Delta tk_3) \]

This is the classic fourth order Runge-Kutta method. The error per step is in the order of \((\Delta t^5)\), but the total accumulated error is in the order of \((\Delta t^4)\). In other words, the Runge-Kutta method is 4\(^{th}\) order in accuracy. It is much more accurate than the Euler method, but we have to do four function evaluation per time step.
Second-order Runge-Kutta method

\[ v(t, x) = x', x(t_i) = x_i \]

\[ k_1 = v(t_i, x_i) \]

\[ k_2 = v \left( t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2} k_1 \right) \]

\[ x_{i+1} = x_i + \Delta t k_2 + O(\Delta t^3) \]

Equivalent to

\[ x_{i+1} = x_i + \frac{\Delta t}{2} (v_i + v_{i+1}) \]
Integration formulas require repeated transformation from global to local coordinates. To move a point through a dataset under the influence of a vector field, we need first to identify the cell that contains the point, next to compute the velocity at that point by interpolating the velocity from the cell points, and then to reposition the point with an increment. The process is repeated until the point exits the dataset or the distance or time traversed exceeds some specified value.

To improve the search procedure, determining the starting location of the particle by a global search, followed by an local incremental search.

Coordinate transformation can be used to speed up the time animation by transform the vector field from global coordinate to local coordinate, do integration in local coordinate, and then transform the selected points in the path back to global space.
Streamlines

- In animation, we can connect the point position \( x(t) \) over many time steps, such that a particle trace is represented by a line. Three line representation schemes are discussed here: particle traces, streaklines, and streamlines. Streamlines, streaklines, and particles are equivalent to one another if the flow is steady.
- Particle traces are trajectories traced by fluid particles over time.
- Streaklines are the set of particle traces at a particular time \( t_i \) that have previously passed through a specified point \( x_i \).
- Streamlines are integral curves along a curve \( s \) satisfying the equation, where \( s = s(x,t) \) for a particular time \( t \).

\[
s = \int_0^t vds
\]
6.4 Tensor algorithms

\[
\begin{bmatrix}
\delta_x & \tau_{xy} & \tau_{xz} \\
\tau_{yx} & \delta_y & \tau_{yz} \\
\tau_{zx} & \tau_{zy} & \delta_z
\end{bmatrix}
\]

\[
\begin{bmatrix}
\frac{\partial u}{\partial x} & \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\
\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \frac{\partial v}{\partial y} & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\
\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) & \frac{\partial w}{\partial z}
\end{bmatrix}
\]

Stress tensor

Stress and strain tensors. Normal stresses in the x-y-z coordinate directions indicated as $\delta_x$, $\delta_y$, $\delta_z$, shear stresses indicated as $\tau_{ij}$. Displacement represented by $u$, $v$, $w$.

In these tensors, the diagonal coefficients are the normal stresses and strains, and the off-diagonal terms are the shear stresses and strains. Normal stress and strain are perpendicular to a specified surface, while the shear stress and strain are tangential to the surface. Normal stress is either compression and tension, depending on the sign of the coefficient.

A 3x3 real symmetric matrix can be characterized by three vectors in 3D called Eigenvectors, and 3 numbers called the eigenvalues. The eigenvectors form a 3D coordinates system whose axes are mutually perpendicular. Both eigenvectors and eigenvalues are very significant in simulation.
Eigenvalues and eigenvectors

Given a matrix $A$, the eigenvector $x$ and eigenvalue $\lambda$ must satisfy the relation:

$$A \cdot x = \lambda x$$

The matrix determinant must satisfy:

$$\det|A - \lambda I| = 0$$

This could be a $n^{th}$ degree polynomial in $\lambda$, whose roots are the eigenvalues. After we obtain the eigenvalues, the eigenvectors can be found by substitute the eigenvalues into $A \cdot x = \lambda x$ to get eigenvectors.

The eigenvectors of the matrix can be written as

$$v_i = \lambda_i \cdot e_i$$

Where $e_i$ is a unit vector in the direction of the eigenvalue, and $\lambda_i$ is the eigenvalues Of the system. The eigenvalues may be orderes as

$$\lambda_1 \geq \lambda_2 \geq \lambda_3$$

Then the corresponding eigenvectors $v_1$, $v_2$, and $v_3$ are the major, medium and minor eigenvectors.
Tensor ellipsoids

- The eigenvectors form a local coordinate system, since the eigenvectors are known to be orthogonal. The axes can be taken as the major, medium and minor axes of an ellipsoid.

- To form the ellipsoid, we begin by positioning a sphere at the tensor location. The sphere is then rotated around its origin using the eigenvectors, and the eigenvalues are used to scale the sphere. The eigenvectors can be directly used to create the rotation matrix, and the point coordinates x-y-z and eigenvalues $\lambda_1$, $\lambda_2$, $\lambda_3$, are inserted into the translation and scaling matrix. Eventually a transformation matrix is formed.

$$T = T_T \cdot T_R \cdot T_S$$
6.5 Modeling algorithms

- Source objects --- to begin the visualization
- Implicit function --- an important class of source objects
- Implicit modeling --- a supplement to implicit function
- Glyphs (icons) --- an object that is affected by its input data
- Cutting --- to cut through a dataset with a surface and then display the interpolated data values on the surface.
Source objects

- Source objects are used to create geometry such as spheres, cones, or cubes to support visualization context or are used to read in data files.

- Modeling simple geometry: real-world objects are often represented by simple geometric representations such as spheres, cones, cubes, and other simple geometric objects.

- Supporting geometry: in the visualization process we may use source objects to create supporting geometry. For streamlines, the points determine the initial positions for generating the streamlines, the probe filter uses the points as the position to compute attribute values such as scalars, vectors, or tensors.

- Data attribute creation: source objects can be used as procedures to create data attributes. For example, we can procedurally create texture coordinates.
Implicit function

- Implicit function is the function of the form: \( F(x, y, z) = c \)

- Properties of implicit function:
  - Simple geometric description. It is a convenient tool to describe common shapes such as planes, spheres, cylinders, cones, ellipsoids, and quadrics.
  - Region separation. It separates 3D Euclidean space into three distinct regions, inside, on, and outside the implicit function.
  - Scalar generation. It converts a position in space into a scalar value. For example, given an implicit function, we can sample it at a point \((x_i, y_i, z_i)\) to generate a scalar value \(c_i\).
Implicit function: modeling objects

- Implicit function can be used alone or in combination to model geometric objects, such as generating isosurface.
- Implicit function can be used to create complex objects using the boolean operators union, intersection, and difference.
- The union between two implicit function is
  \[ F \lor G = \min(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \]
  the intersection between two function is
  \[ F \land G = \max(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \]
  the difference of the two functions is
  \[ F - G = \max(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \]
Implicit functions: selecting data

We can take advantage of the properties of implicit functions to select and cut data.

Select and extract data with an implicit function means choosing cells, points, and associated attribute data that lie within a particular region of the function.

To determine whether a point x-y-z lies within a region, we simply evaluate the point and examine the sign the result. A cell lies in a region if all its points lie in the region.
Implicit functions: visualizing mathematical descriptions

- Some functions are often discrete or probability based. They cannot be written in continuous implicit functions. A method called strange attractor is used to visualize such nonlinear, dynamic, and chaotic systems, in the form of differential equations.

- To visualize the dynamic systems described by differential equations, we need to treat the variables x, y, z as the coordinates of a three-dimensional space, and integrate the equations to generate the system “trajectory”, the state of the system through time. The integration is carried out within a volume and scalars are created by counting the number of times each voxel is visited. By integrating long enough, we can create a volume representing the “surface” of the strange attractor.
Implicit modeling is an extension to implicit function. In implicit modeling the scalars are generated using a distance function instead of the usual implicit function. The distance function is computed as the Euclidean distance to a set of generating primitives such as point, lines, or polygons. Implicit modeling can be used to model complex geometry by using boolean combinations (union, intersection, and difference) of the primitives. In implicit modeling, when isosurfaces are generated, more than one connected surface can results.
Glyphs

- Glyphs are objects of geometry, dataset, or graphical images. Glyphs may orient, scale, translate, deform, or somehow alter the appearance of the object in response to data. Glyphs represent the fundamental result of the visualization process. Glyphs are classified into three categories:
  - Elementary icons: to represent elementary information such as surface normal.
  - Local icons: to represent elementary information plus a local distribution of the values around the spatial domain, such as surface normal vector colored by local curvature.
  - Global icons: to show the structure of the complete dataset, such as isosurface.
The data cutting operation requires two pieces of information: a definition of the surface, and a dataset to cut.

A typical application of cutting is to slice through a dataset with a plane, and color map the scalar data and/or warp the plane according to vector value. Implicit functions can be combined with a contouring algorithm to generate cut surfaces, by generating scalars for each point of each cell of a dataset, and then contour the surface value $F(x,y,z)=0$.

The cutting algorithm. For each cell, function values are generated by evaluating $F(x,y,z)$ for each cell point. If all the points evaluate positive or negative, then the surface does not cut the cell. If the points evaluate positive and negative, then the surface pass through the cell. We can use the cell contouring operation to generate the isosurface $F(x,y,z)=0$. Data attribute values can then be computed by interpolating along cut edges.

Multiple planar cuts can be made by specifying multiple isovalues for the cutting algorithm.
6.6 Implementation in *vtk*

(a) Functional model

(b) Object model

**Source design**: Source objects have no visualization data for input and one or more outputs. To create a source object, inheritance is used to specify the type of dataset that the process object creates for output. *vtkSphereSource* is a concrete source object, which inherits from *vtkSource*, indicating it is a source object, and inherits from *vtkPolyDataSource*, indicating that it creates polygonal data on output. So it creates a polygonal representation of a sphere.
**Filter design:** to create a filter object, inheritance is used to specify the type of input and output. Using `vtkContourFilter` as an example, the superclasses of it are `vtkSource`, `vtkFilter`, `vtkDataSetFilter`, and `vtkDataSetToPolyDataFilter`. `vtkSource` is a superclass of `vtkContourFilter` means that the filter is a source of visualization data. `vtkContourFilter` is a subclass of `vtkFilter` means that `vtkContourFilter` has at least one input. `vtkDataSetFilter` specifies the type of data that `vtkContourFilter` takes as input, i.e., a dataset. In this example, the filter object receives a general dataset as input and creates polygonal data on output.

The class `vtkDataSetFilter` enforces filter input type with the type checking features of the C++ Compiler. It accepts type `vtkDataSet` or its subclasses. Since `vtkDataSet` is a base class for all data types, this filter will accept any type as input. Specialized filters are derived from other classes. Filters that accept polygonal data are derived from `vtkPolyDataFilter`, and filters that accept structured point datasets are derived from `vtkStructuredPointsFilter`.

**Functional model**

**Object model**
Mapper design: mapper objects have one or more inputs and no visualization data output. In vtk two different types of mappers are available, graphics mappers and writers. Graphics mappers interface geometric structure and data attributes to the graphics library, and writers write datasets to disk or other I/O devices. Mappers take datasets as input, and type enforcement is required. Both the vtkPolyDataMapper and vtkSTLWriter classes implement a SetInput() method to enforce the input to be of type vtkPolyData.

The subclasses of vtkMapper must implement the Render() method. This method is exchanged by the graphics system actors and its associated mappers during the rendering process. The purpose of the method is to map its input dataset to the appropriate rendering library/system. Subclasses of the class vtkWriter must implement the WriteData() method. This method causes the writer to write its input dataset to disk (or other I/O device).
Color maps

In **vtk**, color maps are created using instances of the class `vtkLookupTable`. This class allows you to create a lookup table using HSVA specification. `vtkLookupTable` also has the function of loading colors directly into the table through RGBA system.

```cpp
vtkLookupTable *lut = vtkLookupTable::New();
lut->SetHueRange(0.6667, 0.0);
lut->SetSaturationRange(1.0, 1.0);
lut->SetValueRange(1.0, 1.0);
lut->SetAlphaRange(1.0, 1.0);
lut->SetNumberOfColors(256);
lut->Build();
```

```cpp
vtkLookupTable *lut = vtkLookupTable::New();
lut->SetNumberOfColors(3);
lut->Build();
lut->SetTableValue(0, 1.0, 0.0, 0.0, 1.0);
lut->SetTableValue(0, 0.0, 1.0, 0.0, 1.0);
lut->SetTableValue(0, 0.0, 0.0, 1.0, 1.0);
```
Lookup tables in **vtk** are associated with the graphics mappers. Mappers will automatically create a red to blue lookup table if no table is specified. Users can create their own lookup tables using the mapper->SetLookupTable(lut) operation where mapper is an instance of vtkMapper or its subclasses.

Mappers use their lookup table to map scalar values to colors. If no scalars are present, the mappers and their lookup tables do not control the color of the object, and we use the method actor->GetProperty()->SetColor(r,g,b) to specify color, where r, g, and b are floating-point values.

If the users want to prevent scalars from coloring the objects, they can use the method mapper->ScalarVisibilityOff() to turn off color mapping. Then the actor’s color will control the color of the object.

The scalar range is specified with the mapper by the method mapper->SetScalarRange(min, max)

The users can derive their own lookup tables, such as vtkLogLookupTable(), which performs logarithmic mapping of scalar values to table entries.
Implicit functions

The subclasses of `vtkImplicitFunction` include `vtkPlane`, `vtkSphere`, `vtkCylinder`, `vtkCone`, `vtkQuadric`, and `vtkImplicitBoolean`. The class `vtkImplicitBoolean` allows the users to create boolean combinations of these implicit function primitives.

The subclasses of `vtkImplicitFunction` must implement the two methods, `Evaluate()` and `Gradient()`. The method `Evaluate()` returns the value of the function at point \((x,y,z)\), and the method of `Gradient()` returns the gradient vector to the function at point \((x,y,z)\).
Contouring

- Scalar contouring is implemented in **vtk** with **vtkContourFilter**, which accepts any dataset type as input. A contour table is associated with each cell type, so each cell will generate appropriate contouring primitives. A tetrahedron cell type implements *marching tetrahedron*, and creates triangle primitives, and the triangle cell type implement *marching triangles* and generates line segments.

- There are two kinds of contour filters, **vtkContourFilter**, and **vtkMarchingCubes**. **vtkContourFilter** is a general filter, which generates point, line, and surface contouring primitives. **vtkMarchingCubes** is specific to the dataset type structured points.
The performance of specific filter generation and general filter generation is compared. The vtkMarchingCubes outperforms vtkContourFilter, and the difference is large for low resolution, i.e. small datasets. This is because the total number of voxels increases as the resolution cubed, while the voxels containing the isosurface increase as the resolution squared. So when the datasets become larger, more voxels are empty and are not processed.

<table>
<thead>
<tr>
<th>resolution</th>
<th>Specific (with normals)</th>
<th>General (no normals)</th>
<th>Factor</th>
<th>General (with normals)</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>64×64×93</td>
<td>1.000</td>
<td>2.889</td>
<td>2.889</td>
<td>7.131</td>
<td>7.131</td>
</tr>
<tr>
<td>128×128×93</td>
<td>5.058</td>
<td>11.810</td>
<td>2.330</td>
<td>23.260</td>
<td>4.60</td>
</tr>
<tr>
<td>256×256×93</td>
<td>37.169</td>
<td>51.620</td>
<td>1.390</td>
<td>87.230</td>
<td>2.350</td>
</tr>
</tbody>
</table>
// define implicit function
vtkQuadric *quadric=vtkQuadric::New();
quadric->SetCoefficients(0.5, 1, 0.2, 0, 0.1, 0, 0, 0.2, 0, 0);

vtkSampleFunction *sample=vtkSampleFunction::New();
sample->SetSampleDimensions(50, 50, 50);
sample->SetImplicitFunction(quadric);

vtkContourFilter *contour=vtkContourFilter::New();
contour->SetInput(sample->GetOutput());
contour->GenerateValues(5, 0, 1.2);

vtkPolyDataMapper *contourMapper=vtkPolyDataMapper::New();
contourMapper->SetInput(contour->GetOutput());
contourMapper->SetScalarRange(0, 1.2);

vtkActor *contourActor=vtkActor::New();
contourActor->SetMapper(contourMapper);

// create outline
vtkOutlineFilter *outline=vtkOutlineFilter::New();
outline->SetInput(sample->GetOutput());

vtkPolyDataMapper *outlineMapper=vtkPolyDataMapper::New();
outlineMapper->SetInput(outline->GetOutput());

vtkActor *outlineActor=vtkActor::New();
outlineActor->SetMapper(outlineMapper);
outlineActor->GetProperty()->SetColor(0,0,0);

Contouring quadric function
vtkCutter performs cutting of all vtk cell types. vtkCutter requires an implicit function that will be evaluated at each point in the dataset. Then each cell is cut using the cell’s Contour method. Any point attributes are interpolated to the resulting cut vertices. The sorting order for the generated polygonal data can be controlled with the SortBy method. The default sorting, SortByValue(), processes cells in the inner loop for each contour value. SortByCell() processes the cutting value in the inner loop and produces polygonal data that is suitable for back-to-front rendering.

vtkCutter uses an implicit function to calculate scalar values but vtkContourFilter uses the scalar data associated with the dataset’s point data.

The SetValue() and GenerateValues() methods permit the user to specify which multiple scalar values to use for cutting.
Glyphs is implemented in \texttt{vtk} as \texttt{vtkGlyph3D}, which is an example of an object that takes multiple inputs. One input specified by \texttt{SetInput()} method defines a set of Points and possible attribute data at those points. The second input specified by the \texttt{SetSource()} method defines a geometry to be copied to every point in the input dataset. The source of \texttt{vtkGlyph3D} is of type \texttt{vtkPolyData}. So any filter or sequence Filters that create polygonal data, or a polygonal dataset may be used to describe the glyph's geometry.
Streamlines and particle motion require numerical integration to guide a point through the vector field. It is implemented in vtk by the base class `vtkStreamer`, which is responsible for generating a particle path through a vector field of specified length. Each subclass of `vtkStreamer` takes advantage of this capability to move through a vector field but implemented its own particular representational technique to depict particle motion. `vtkStreamLine` draws connected lines while particle motion is shown by combining the output of `vtkStreamPoints` with the `vtkGlyph3D` object. We can place spheres or oriented objects such as cones or arrows at points on the particle path created by `vtkStreamPoints`. 
Attribute transformations create or modify data attributes without changing the topology or geometry of a dataset. Therefore filters that implement attribute transformation, such as vtkElevationFilter, can accept any dataset type as input, and may generate any dataset type as output. Because filters must specify the particular type of data they output, filters cannot create general dataset types on output, since the type vtkDataSet is an abstract type and must be specialized to create an instance.

The problem can be overcome by using the virtual constructor MakeObject(). We can construct filters that output abstract data types like vtkDataSet. This is done by applying MakeObject() to the input of the filter, then return a pointer to a concrete object that is the output of the filter. The result is a general filter object that can accept any dataset type for input and creates the general vtkDataSet type as output.

In vtk, it is implemented in the abstract class vtkDataSetToDataSetFilter.
Visualization of blood flow in the human carotid arteries, where cone glyphs are used to indicate flow direction and magnitude.

```python
vtkStructuredPointsReader reader
reader SetFileName "../..../vtkdata/carotid.vtk"
reader DebugOn
vtkThresholdPoints threshold
threshold SetInput [reader GetOutput]
threshold ThresholdByUpper 200
vtkMaskPoints mask
mask SetInput [Threshold GetOutput]
mask SetOnRatio 10
vtkConeSource cone
cone SetResolution 3
cone SetHeight 1
cone SetRadius 0.25
vtkGlyph3D cones
cones SetInput [mask GetOutput]
cones SetSource [cone GetOutput]
cones SetScaleFactor 0.005
cones SetScaleModeToScaleByVector
vtkPolyDataMapper vecMapper
vecMapper SetInput [cones GetOutput]
vecMapper SetScalarRange 2 10
```
vtkStructuredPointsReader reader
reader SetFileName “../../../vtkdata/carotid.vtk”
reader DebugOn

vtkPointSource source
source SetNumberOfPoints 25
source SetCenter 133.1 116.3 5.0
source SetRadius 2.0

vtkThresholdPoints threshold
threshold SetInput [reader GetOutput]
threshold ThresholdByUpper 275

vtkStreamLine streamers
streamers SetInput [reader GetOutput]
streamers SetSource [source GetOutput]
streamers SetMaxmumPropagationTime 100.0
streamers SetIntegrationStepLength 0.2
streamers SpeedScalarsOn
streamers SetTerminalSpeed 0.1

vtkTubeFilter tubes
tubes SetInput [streamers GetOutput]
tubes SetRadius 0.3
tubes SetNumberOfSides 6
tubes SetVaryRadiusToVaryRadiusOff

vtkPolyDataMapper streamerMapper
streamerMapper SetInput [tubes GetOutput]
streamerMapper SetScalarRange 2 10

**Visualization of blood flow in the human carotid arteries, where streamtubes are used to visualize flow vectors**